Reconciling Compiler Optimizations and WCET Estimation Using Iterative Compilation

Mickaël Dardaillon, Stefanos Skalistis, Isabelle Puaut, Steven Derrien

Reconciling Compiler Optimizations and WCET Estimation Using Iterative Compilation

Mickaël Dardaillon*, Stefanos Skalistis†, Isabelle Puaut‡, Steven Derrien§
*INSA, IETR, UMR 6164, F-35708 RENNES
†‡§Univ Rennes, Inria, CNRS, IRISA, F-35708 RENNES
*mickael.dardaillon@insa-rennes.fr, †stefanos.skalistis@inria.fr, ‡isabelle.puaut@irisa.fr, §steven.derrien@irisa.fr

***Abstract*—Static Worst-Case Execution Time (WCET) estimation techniques operate upon the binary code of a program in order to provide the necessary input for schedulability analysis techniques. Compilers used to generate this binary code include tens of optimizations, that can radically change the flow information of the program. Such information is hard to be maintained across optimization passes and may render automatic extraction of important flow information, such as loop bounds, impossible. Thus, compiler optimizations, especially the sophisticated optimizations of mainstream compilers, are typically avoided. In this work, we explore for the first time iterative-compilation techniques that reconcile compiler optimizations and static WCET estimation. We propose a novel learning technique that selects sequences of optimizations that minimize the WCET estimate of a given program. We experimentally evaluate the proposed technique using an industrial WCET estimation tool (AbsInt aiT) over a set of 46 benchmarks from four different benchmarks suites, including reference WCET benchmark applications, image processing kernels and telecommunication applications. Experimental results show that WCET estimates are reduced on average by 20.3% using the proposed technique, as compared to the best compiler optimization level applicable.**

***Index Terms*—Worst-Case Execution Time Estimation, Compiler optimizations, Iterative Compilation**

I. Introduction

Safe upper bounds of Worst-Case Execution Times (WCET) are the key input of schedulability analysis, that ensures that deadlines of hard real-time tasks are met in all situations, including the worst case. WCET calculation techniques must be *safe* (provide upper bounds of execution times) and as *tight* as possible (provide a bound as close as possible to the actual WCET, which is in general unknown).

Many WCET estimation techniques have been developed in the past [1], using either *static analysis techniques* or *measurement-based techniques*. Static WCET analysis techniques do not execute the code, and operate on the structure of the binary code. A *hardware-level* analysis step first calculates the worst-case execution time of sequences of instructions (basic blocks); this step has to be performed on the binary code, because the timing of instructions within basic blocks depends on the actual instructions generated by the compiler, as well as the mapping of instructions to memory addresses. A *calculation phase* then computes the WCET estimate of the entire program from the WCET of its basic blocks [2]. On the other hand, measurement-based techniques use end-to-end measurements to estimate WCETs. Contrary to measurement-based techniques, static WCET estimation techniques, used in this work, are guaranteed to provide safe upper bounds of WCETs, provided that they are able to build a safe model of the target hardware.

Static WCET estimation techniques require that all paths in programs are of bounded length, meaning the maximum number of iterations of loops, and the maximum depth of recursion are both bounded. Flow information (loop bounds, bounds on recursion or more elaborate flow information such as mutually exclusive paths) may be obtained using static analysis, or provided by the user as annotations. In general, automatic extraction of flow information using static analysis is to be preferred to annotations, that are error-prone.

Compilers translate high-level languages into binary code, fit for microprocessors. Modern compilers typically apply tens of optimizations (loop transformations, dead code elimination, factorization of redundant code, function inlining, etc.) to deliver more performance. While optimizations are key in delivering performance for the average case, optimizations are not always beneficial for the WCET. In addition, some optimizations may radically alter the program control flow and, thus, may break the automatic determination of flow information (such as loop bounds). The general problem when optimizing code in compilers, either for average or worst-case execution time, is that there is no precise processor performance model in the compiler because of the processor complexity. Additionally, optimization passes are program-specific and interdependent, in that they may have additive or destructive impact dependent on the optimization combination and the program to optimize. This prevents the use of exact optimization techniques such as Integer Linear Programming (ILP) or dynamic programming, and motivates research in the field of iterative compilation.

In this paper, we propose for the first time a technique to reconcile compiler optimizations (to have as good as possible performance, in average case and worst case) and WCET estimation, using mainstream compilers. Inspired from iterative compilation, commonly used to optimize average-case performance [3], [4], we propose a technique for an automatic selection of optimization passes that both allows (i) the automatic extraction of precise flow information; (ii) optimization of WCETs. Since the space of optimizations to be explored is huge, techniques for selecting sequences of optimization passes have to find good candidate sequences in reasonable amount of time. More precisely, our contribution is

an exploration technique for selecting optimization sequences tailored to a specific program, which improves analyzability and reduces the program's WCET estimate, compared to standard optimization levels (*O0* to *O3*). Experimental results show that the proposed exploration technique results in lower WCET estimates than using random or genetic selection of optimization sequences.

The contributions of this research are the following:

- We propose a technique for automatic selection of optimization passes that allows (i) automatic WCET analysis, in particular automatic flow fact extraction, and (ii) WCET estimates minimization compared to standard optimization levels (*O0* to *O3*). The optimization sequences selection stage uses a machine learning technique that exploits correlations between optimization passes. In contrast to most research on iterative compilation, which focuses on average-case performance, our exploration technique optimizes analyzability and worst-case performance.
- We provide a detailed experimental evaluation of the proposed technique on code for a LEON3 processor. Experiments are performed using the mainstream compiler *LLVM* [5] and the aiT industrial timing analysis tool [6]. Experimental results show that the WCET estimates of a large selection of benchmarks (46 benchmarks from the benchmark suites Mälardalen, Polybench, MiBench and PolyMage) are reduced by an average of 20.3%, as compared to the best compiler optimization level (*O0* to *O3*) applicable without failure. Thanks to the very general nature of our approach, this technique can be implemented in industry realistically.
- As a side result, our research also provides a technique to select the best *unrolling factor* parameter for the aiT industrial static WCET estimation tool. The *unrolling factor* is a parameter of aiT that allows to trade the quality of WCET estimates (higher unrolling factor results in better WCET estimates) against analysis time (higher unrolling factor results in higher analysis time).

The rest of this paper is organized as follows. Section II further motivates this research, by highlighting the difficulty of selecting optimization sequences that optimize WCET estimates. Section III then presents the proposed method for selecting optimization passes. Section IV evaluates the quality of our proposal on a large set of benchmarks. Section V then presents the automatic selection of the aiT's unrolling factor parameter. Our research is compared to related work in Section VI. Concluding remarks and directions for future research are given in Section VII.

II. Motivation

Mainstream compilers provide standard code optimization levels (from *O0*, no optimization, to *O3*, highly optimized code), and also provide ways manually select optimization sequences, defined as an ordered list of optimization passes in which repetitions are allowed. This section motivates our research on WCET-oriented exploration for the best optimization sequence, through observations on examples. The experimental setup used in this section is the same used for the experimental evaluation of our work, i.e. use of the aiT WCET estimation tool for the LEON3 processor (see Section IV). All observations are made by randomly selecting 1000 optimization sequences of size 10 to 80, on two benchmarks (*des* and *harris*), with the *virtual unrolling factor* of the aiT timing analysis tool of 2 and 4 respectively.

Timing analysis may fail on optimized code

Figure 1 shows the number of *failures* encountered during the experiments. Failures encompass compiler failures (compiler internal errors, for example due to unsupported sequences of optimizations) and WCET analysis errors (mainly incapability of the tool to derive flow information, more marginally memory exhaustion or very long analysis times).

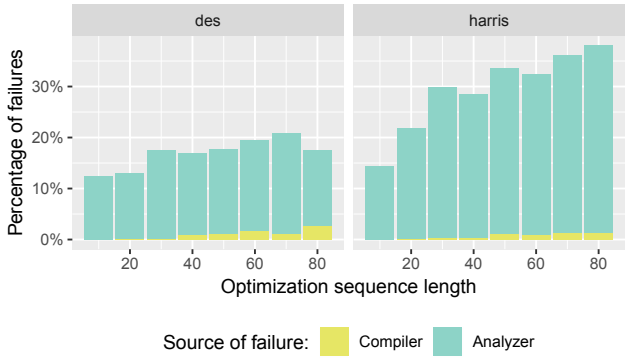

Fig. 1. Percentage of compiler and analyzer failures for 1000 random generated sequences of size 10 to 80.

The figure shows a significant number of analysis failures, outlining that some optimizations, either in isolation or in conjunction with other optimizations, hinder temporal analysis of the code. Obviously, the optimization sequences resulting in failures should not be selected. The presence of analysis failures is specific to WCET-directed search for compiler optimizations, as compared to iterative compilation techniques optimizing average-case performance.

Optimization sequences may result in WCET estimates better than O3 or worse than O0

As already observed for average-case performance [3], Figure 2 shows that some optimization sequences result in WCETs that are lower than when using optimization level *O3*. For the *des* benchmark, 20% of the observations provide better estimated WCETs than *O3*; for *Harris*, 14% of the observations are better than *O1*, and levels *O2* and *O3* make the WCET analysis fail.

Figure 2 also shows that a significant number of optimization sequences result in WCETs estimates that are much larger (up to a factor of two for *harris*) than when compiling at *O0*. In addition, we observed that a significant amount of sequences have a negligible impact on the WCET estimate compared to

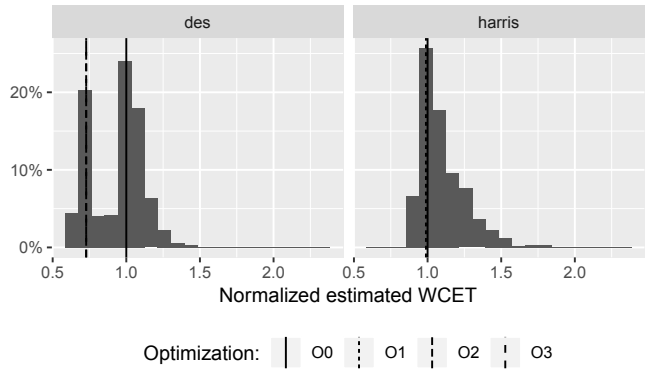

Fig. 2. Distribution of WCET estimates on 1000 random generated sequences normalized to the estimated WCET at *O0*, excluding failed analyses.

O0 (18% for *des*, 16% for *harris* result in WCET estimates that are within 2% of WCET estimates at *O0*).

Optimizations are application specific

Figure 3 shows the distribution in the number of activated optimization passes for the best 100 WCET estimates. The results demonstrate that the number of optimization passes, that yields low WCET estimates, varies among benchmarks. More optimizations have to be activated for *des* than for *Harris*.

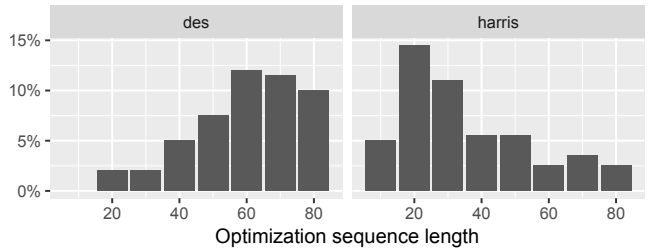

Fig. 3. Distribution of the best 100 estimated WCET for 1000 random generated sequences of size 10 to 80.

The actual optimization sequences (i.e. the number of optimizations passes, their type and their particular ordering), yielding low WCET estimates are also benchmark dependent, as illustrated in Table I, that shows the sequence with the best WCET for *des* and *Harris*.

Finally, WCET estimation tools such as aiT use parameters that enable a trade-off between analysis precision vs analysis time. The *virtual unrolling* parameter is one of such parameters. As further detailed in Section V, larger unrolling factors improve the precision of WCET analysis at the cost of a longer analysis runtime. The best value for this parameter is also benchmark-dependent (*2* for *des* and *4* for *Harris*), and therefore has to be part of the search.

Compared to iterative compilation techniques for average-case performance, our work (i) optimizes a different metric (WCET estimate instead of average-case performance), (ii) disregards optimization sequences that hinder analysis, (iii) not only selects the best optimization sequences but also the best parameters of the timing analysis tool.

TABLE I
BEST OPTIMIZATION SEQUENCE FROM 1000 RANDOM GENERATED SEQUENCES OF SIZE 1 TO 50

des	-loop-rotate -sink -indvars -partial-inliner -loop-simplify -always-inline -deadargelim -mem2reg -loop-unroll -partial-inliner -lcssa -simplifycfg -lowerinvoke -inline -loop-reduce -loop-simplify -strip-dead-prototypes -loop-unswitch -loop-deletion -die -reassociate -constmerge -loop-deletion -lowerinvoke -ipconstprop -globalopt -mergereturn
harris	-always-inline -always-inline -functionattrs -prune-eh -loop-simplify -dce -strip-dead-debug-info -codegenprepare -mem2reg -loop-rotate -partial-inliner -simplifycfg -functionattrs -adce -prune-eh -globaldce -loop-unswitch -strip-nondebug -strip-nondebug -simplifycfg -functionattrs -loweratomic -constmerge -globaldce -strip-nondebug -globaldce -constprop -loop-deletion -lcssa -instcombine

III. SELECTION OF OPTIMIZATION SEQUENCES

As highlighted in Section II, selecting the sequence of optimization passes that improves WCET estimates for a given program is rather challenging. This stems from intricate correlations among optimization passes, with unpredictable impact on the estimated WCET. In addition, sequences can be arbitrarily long, thus the space of optimization sequences is vast, that is practically impossible to fully explore.

In order to reduce the explored space and improve WCET estimates within reasonable time, our approach focuses on *characterizing* the impact (*positive*, *neutral* or *negative*) of the individual passes of a sequence with respect to already evaluated sequences. By construction, the characterization strategy classifies as *neutral* and *negative* the passes that frequently exhibit that behavior after a significant number of attempts. The main purpose of the characterization is to avoid passes that frequently exhibit a *neutral* or *negative* impact, as such passes are not likely to improve the estimated WCET. Conversely, passes that have been observed to frequently exhibit a *positive* impact, or passes that have not been applied often enough to establish their impact, are considered beneficial.

After having characterized the individual passes of each evaluated sequence, we apply a *weighting* scheme to guide the selection of passes towards those having frequently a *positive* impact, and away from passes that have frequently a *neutral* impact. The weighting scheme is inspired by association rule learning of the *data mining* domain [7]. Association rule learning identifies frequent causality relations among subsets within transactions of a database. In a similar manner, our characterization strategy and weighting scheme identifies frequent causality relations between sets of optimization passes and impact on the resulting WCET. We opted for a weighting scheme, instead of an established algorithm from the domain, as such learning to be successful requires a large dataset. Acquiring such a large dataset is not only computationally expensive, which is not in line with our motivation, but

also is not reusable across programs, since th optimization passes is program-specific.

The proposed approach, illustrated in Figur typical iterative compilation workflow. Given high-level language and a compiler for the targ the approach starts by compiling the given no optimization (*O0*) and follows by analyz WCET estimation tool. This establishes a basel the program. Subsequently, a new optimizatic generated, by selecting (i) its length randomly pass of the sequence in a weighted random mar to the proposed *weighting* scheme (weights are i The program is then recompiled and analyzed, new WCET estimate. The newly generated se compared to the previously evaluated sequences common passes, and the impact of its passes According to this new characterization of passe, the weights are updated and the process is repeated.

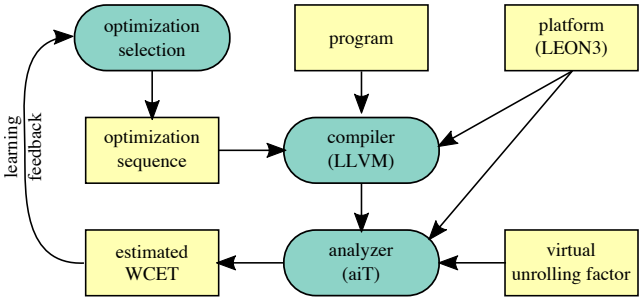

Fig. 4. Iterative compilation workflow

A. Characterization of optimization passes

Discovering the impact of passes (*positive*, *neutral* or *negative*) is not straightforward, as there is not enough information to individually evaluate which passes contribute positively/negatively to the estimated WCET. For example, consider two sequences, ABC and ABD that have improved WCET (compared to *O0*). Let their WCET estimates have the following ordering $\text{WCET}_{ABD} < \text{WCET}_{ABC}$ (lower is better). It is hard to decide, whether optimization pass D is responsible for the improved WCET estimates of ABD or optimization pass A has a negative impact on ABC.

To establish the impact of a sequence of passes with respect to another sequence, the proposed characterization is founded upon the lattice of their *sets*[1] of passes (the partial order in the lattice is set inclusion, see Figure 5). The characterization strategy for a given sequence starts by finding the largest subset that has been already evaluated. The comparison of their WCET estimates characterizes the given sequence as *positive*, *neutral* or *negative*. This characterization is applied to their non-common passes, while the common passes inherit the characterization from the largest subset. For example (Figure 5), if sequence AB is evaluated as *positive* compared

[1]A sequence of passes is denoted as ABC and the corresponding set of passes is denoted as $\{A, B, C\}$

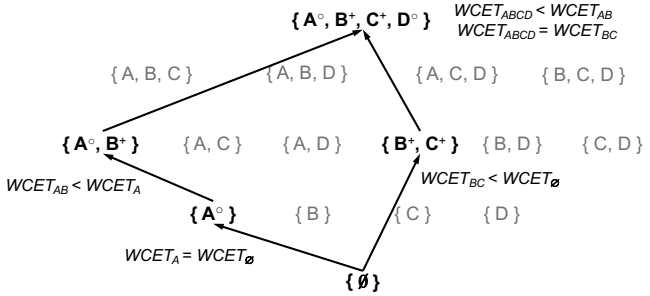

Fig. 5. Example of the characterization strategy for four optimization passes. Bold sets have been evaluated, while grey have not. Optimization pass X is denoted as X^+, X°,X^- when characterized as *positive*, *neutral* or *negative*, respectively. For each arrow (subset to superset ordering in the lattice), the corresponding variation of estimated WCET is shown (lower is better).

In case there are more than one largest subset for a given sequence, then (i) the non-common passes among all the sets and the given sequence, are characterized according to the sequence characterization, (ii) the non-common passes among the sets retain their characterization of their set and (iii) the common passes retain the most frequent characterization among the sets, with ties being resolved by the characterization of the sequence. Continuing our previous example, consider the sequence $ABCD$ is evaluated as positive compared to $\{A, B\}$ and as neutral compared to $\{B, C\}$; the sequence is characterized as neutral, its WCET is equal to WCET_{BC}. Pass D is characterized as neutral as it is not present in any of the subsets. Passes A and C retain the characterization of their corresponding set (as the non-common part among the largest subsets) and pass B retains its characterization since it is positive in both sets.

B. Weighting scheme

The objective of the weighting scheme is to avoid passes that have frequently been observed to be *neutral* and promote passes that either have been frequently positive or have not been tried many times. This is achieved by setting the weight of each $pass$ to:

$$W_{pass} = \frac{N^+_{pass}}{\max\left(1, N^\circ_{pass}\right)} \times 85\% + 15\% \qquad (1)$$

where, N^+_{pass} is the number of times it has been characterized as *positive* among evaluated sequences, and N°_{pass} is the number of times it has been characterized as *neutral*. In order to avoid convergence to local optima, we add a fixed 15%, determined experimentally, to maintain some randomness. Also note that while our weighting scheme does not include a term for negative passes, passes with frequently negative impact are avoided, since the weights are normalized in the weighted random selection.

C. Cleaning of optimization passes

In order to enhance the detection of *neutral* passes, we use an additional *cleaning* phase. This method is largely used in

iterative compilation [3], [8], and its impact in WCET estimation is outlined in the experimental evaluation in Section IV. The cleaning phase, after a sequence has been evaluated, generates a new sequence by removing one pass from the previous sequence. If the resulting sequence has the same or higher WCET estimate as the original one, the pass is removed for all subsequent sequences of the cleaning phase, otherwise it is kept. The cleaning phase continues to the next optimization pass, until all optimization passes have been selected.

Note that all the sequences generated and evaluated by the cleaning phase are also characterized. In this manner, not only *neutral* passes are identified, but also the confidence of *positive* (or *negative*) passes is re-enforced. Cleaning gives a precise characterization of each optimization, at the expense of a large runtime overhead, given that WCET analysis has to be run for every optimization in the sequence. For this reason, the cleaning phase is not applied to all optimization sequences, as further explained in Section IV.

IV. Experimental Evaluation

A. Experimental setup

A standard workflow for iterative compilation is used to explore optimization influence, adapted to WCET estimation, as illustrated previously in Figure 4. The platform targeted for the experimentation is a 32-bit RISC LEON3 processor (compatible with SPARC V8 instruction set) with separate 16 KB 2-way instruction and data caches. Programs are compiled using LLVM 4.0 for the SPARC architecture, using Gaisler BCC 2.0.5 implementation [9]. The LLVM optimizer *opt* was compiled separately, and all its 53 optimizations passes are used in the experiments. Timing analysis is carried out by AbsInt aiT [6] version 18.04 targeting the LEON3 architecture. aiT implements value analysis using abstract interpretation, to determine the range of values in registers, as well as static analysis of the processor micro-architecture and automatic detection of loop bounds [10]. The virtual unrolling factor for each benchmark is chosen as described is Section V. All experiments were conducted inside a virtual machine running Ubuntu version 18.04, on a MacBookPro with an Intel i7-7660U processor and 16 GB of RAM.

By its non standard use of compiler and analysis tool, iterative compilation tends to find corner cases causing bugs during evaluation. To circumvent those bugs the workflow includes timeouts of 60 seconds for LLVM and 120 seconds for aiT. Failures in compiler were similar to those observed in iterative compilation for average case, and accounted for less than 1% of the evaluated sequences. Failures in analysis tool were mostly due to the timing analysis requiring more time than the given timeout, or more rarely due to memory exhaustion from large virtual unrolling factor. Timeout failures were considered as a failure to obtain an estimated WCET and treated as such in the rest of the workflow.

Evaluation of the optimization selection is supported by a large selection of benchmarks from different domains: Mälardalen benchmark suite, that are reference benchmarks for WCET estimation [11]; Polybench benchmarks [12] for linear algebra, physics and statistics; MiBench [13] benchmarks for its applications in security and telecommunications; and PolyMage [14] for image processing. As this work focuses on optimization, we paid special attention to include complex kernels, which can be challenging for WCET estimation.

All benchmarks were refactored into three parts: initialization, computation and result transmission. This separation supports the objective to focus on computation optimization, with analysis carried only on the computation part. Using the computation output with result transmission is essential, otherwise the compiler could consider parts of the computation as dead code and remove them.

As part of this separation, *Mälardalen* benchmarks *bs*, *cover*, *lcdnum* and *ud* were removed as they do not produce output values. *Polybench* was the easiest to port, as it was already structured in this way. *MiBench* provides a large selection of benchmarks used in embedded systems, but many of them access the file system during their execution which limits analysis possibilities. In order to increase the number of large applications from different application domains, several benchmarks were heavily modified to be analyzable. The list of all benchmarks used is presented in Table II, with their estimated WCET at *O0*, using virtual unrolling as defined in Section V.

Four methods were used for the selection of optimization sequences, with two reference methods: *random* and *genetic*, and the two newly introduced methods *association* and *cleaning* (shortcut for association combined with cleaning). All methods use 1000 different optimization sequences for evaluation (duplicates are removed).

a) Random: This method uses sequences of random size from 1 to 50, and each optimization pass is then randomly selected.

b) Genetic: This method follows the main structure of genetic algorithms [15]. It uses generations of 200 individuals, each individual being an optimization sequence, and each gene encoding an optimization. The first generation is created using the random method previously presented. At each generation, half of the population is kept based on the estimated WCET of its individuals. The population is then doubled using crossover, whereby two random individuals are selected and duplicated, their duplicates are split at a random position and their genes are exchanged after the split point. Mutation is applied randomly to 15% of the individuals as a change to a single optimization pass at a randomly selected position in the sequence. Estimation is ran for as many generations as needed until 1000 different individuals are evaluated.

c) Association: This method is based on the association rule learning presented in Section III. It starts with 200 sequences generated using the random method. Optimizations are characterized with the proposed strategy, and weights are derived for each optimization as defined in Section III-B. The next 800 sequences are generated using a weighted random generator to select the optimizations. The model and the weights are updated after each analysis.

TABLE II
BENCHMARKS USED IN THE EXPERIMENTAL EVALUATION

Collection	Benchmark	Estimated WCET at *O0* (cycles)
Mälardalen	cnt	48885
	crc	287296
	des	1143962
	duff	5081
	expint	244327
	fdct	25826
	fir	85578129
	jfdctint	21235
	ludcmp	25305
	matmult	737810
	ns	60809
	nsichneu	166248
	qurt	33168
	sqrt	8611
	statemate	29660
Polybench	2mm	1262084
	3mm	1988631
	adi	6290978
	atax	300561
	bicg	252896
	covariance	1307156
	doitgen	1216712
	durbin	187702
	fdtd-2d	4469129
	floyd-warshall	82188104
	gemm	1475502
	gemver	528009
	gesummv	151566
	heat-3d	3266377
	jacobi-1d	91361
	jacobi-2d	4982515
	lu	2422376
	mvt	304227
	nussinov	53768579
	seidel-2d	7208288
	symm	949889
	syr2k	1162729
	syrk	956099
	trisolv	88809
	trmm	547295
MiBench	CRC32	7005747235
	FFT	20046629
	adpcm_decoder	1002950581
	rijndael	866798
PolyMage	harris	892512876
	pyramid_blend	30415746437

d) Cleaning: This method is an extension of the association method. Each time a new best result is found, the sequence is cleaned as defined in Section III-C to characterize each of its optimization passes precisely. This cleaning is applied to each new best result after the first 200 sequences only.

B. Experimental results

1) Global analysis of results: Table III presents the geometric mean of the best estimated WCET per benchmark for each method. All results are normalized to the best estimated WCET using the four standard optimization levels (*O0* to *O3*). Geometric mean is used due to the large variation between the normalized estimated WCET to give a representative number. From these results the random method is the worst with a 16.8% improvement, and genetic third with an 17.6% improvement. Association is second with a 19.4% improvement, and cleaning brings an additional 1% to reach 20.3% of improvement over the best standard optimization level.

TABLE III
GEOMETRIC MEAN OF THE BEST ESTIMATED WCET PER BENCHMARK, NORMALIZED TO THE BEST STANDARD OPTIMIZATION (*O0* TO *O3*)

Method	Normalized estimated WCET
Random	0.832
Genetic	0.824
Association	0.806
Cleaning	0.797

Although there are differences in improvement for the different methods, the results motivate the use of iterative compilation in the domain of real-time systems.

The best estimated WCET per method, normalized to the best standard optimization level is detailed in Figure 6 for all benchmarks. The benchmarks present very different characteristics, with significant variations in estimated WCET as already presented in Table II, but also in sensitivity to optimizations. The worst acquired result from all four methods is an absence of improvement for benchmark *3mm*, and less than 1% improvement for *jacobi-1d, jfdctint* and *rijndael*. The best improvement in estimated WCET comes to *doitgen* with a 74% improvement.

2) Analysis per method: In order to compare the four methods we ranked the best estimated WCET for each benchmark between the methods. Ties in estimated WCET were solved by giving all the methods in the tie the same ranking. The results of this ranking are presented in Figure 7.

The random method is clearly dominated by all the other methods, but still finds 17% of the best results. Genetic is leading with more than 43% of the best results. Association and cleaning methods are in between with 30% and 39% of the best results respectively. Both methods share a very similar algorithm, which means they explore a similar space, thus splitting the results between them. This is confirmed by observing the cumulative best and second best results, where cleaning gets the best results with 78%, followed by association with 70% and genetic with 67%.

Comparing the ranking with the geometric mean results on Table III, we observe that genetic is leading in terms of best results, but dominated on average by the association and cleaning methods. This highlights a difference between these methods, with genetic being more hit or miss, whereas others are more consistent.

Figure 8 presents the percentage of failures for each method during compilation and analysis. It highlights the particularities of each algorithm related to failures.

In the case of genetic, every failure will be filtered out at the next generation, resulting in fewer failures overall. In

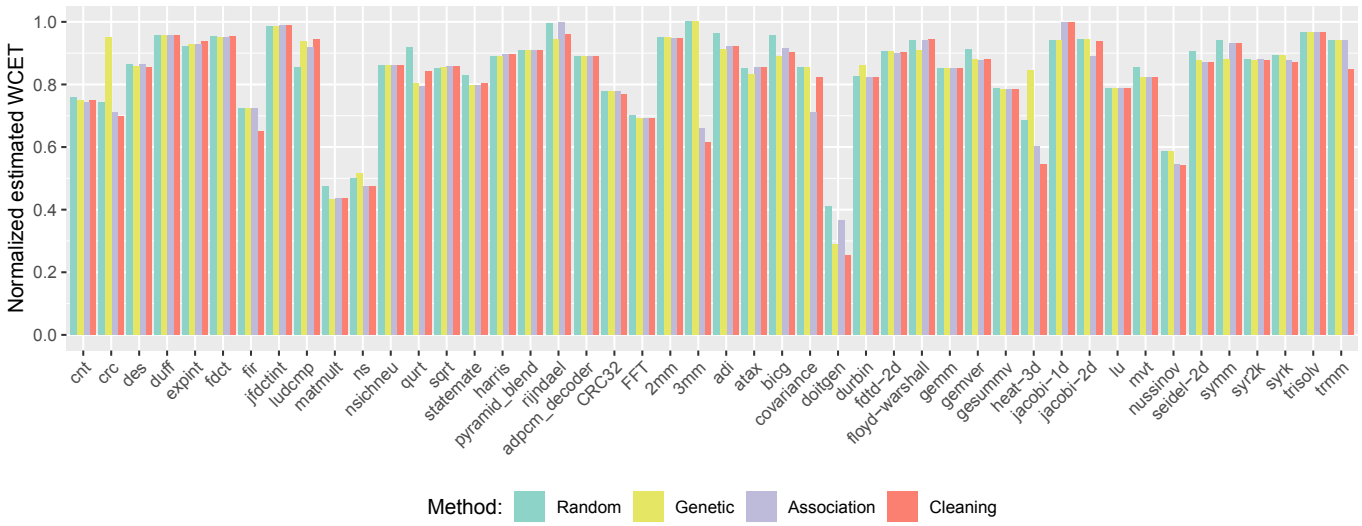

Fig. 6. Best estimated WCET per method normalized to the best standard optimization (*O0* to *O3*)

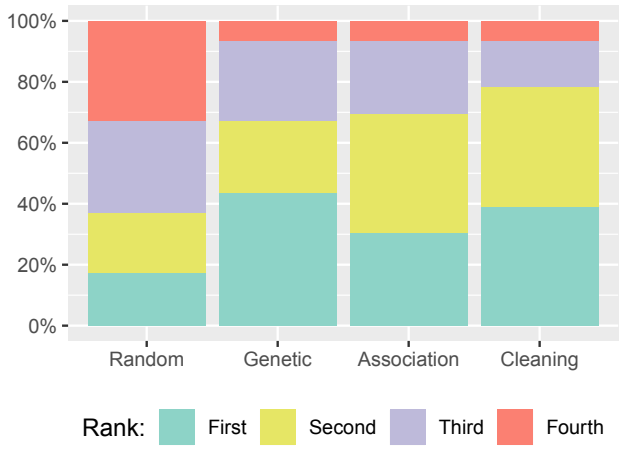

Fig. 7. Ranking of the best estimated WCET for each benchmark between all methods

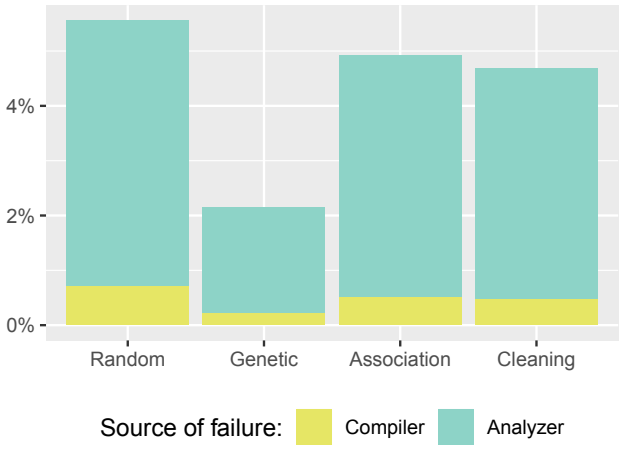

Fig. 8. Percentage of failure across all benchmarks for each method

the case of association and cleaning, the chosen weighting does not use the negative characterization information, which means we get nearly as many failures as a random exploration. One thing to note is that a single addition or removal of an optimization pass can turn the best sequence into a failure, which is unique to WCET estimation. This observation was established during cleaning of the best sequences, where the removal of optimizations led to one compiler and two analyzer failures, starting from the best result. As such, failure is not a good indicator for finding the best results.

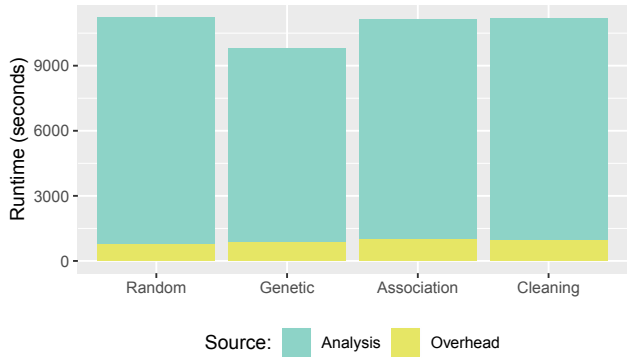

Fig. 9. Geometric mean of runtime for 1000 analyses per method.

Evaluation of runtime per method requires particular care given the inherent variability of running on top of an operating system, inside a virtual machine. We use two measurements for runtime, with timestamps at the beginning and the end of a method to measure *total runtime*, and the runtime reported by aiT per analysis for *analysis runtime*. Both measurements are reported in seconds. To obtain reliable results we removed benchmarks with a total runtime of less than 2000 seconds, meaning less than 2 seconds per analysis. Figure 9 presents the geometric mean of the runtime across benchmarks for each

method. Geometric mean was chosen to compute a meaningful runtime given the large variability across benchmarks (from 360 seconds for *sqrt*, which is excluded, to 34082 seconds for *fdtd-2d*).

The total runtime is divided into *analysis runtime*, as reported by aiT, and *overhead runtime*, as the total minus analysis runtime. We observe that runtime for all methods is dominated by the WCET analysis runtime. The overhead runtime is similar for all methods. Genetic total runtime is smaller than the others, which is correlated to the number of failures presented previously. This can be explained by the longer runtime resulting from failures caused by timeouts.

Figure 10 presents the distribution of lengths for the best sequence of each benchmark across all four methods, before and after cleaning the sequence. Specifically, each sequence and associated benchmark was evaluated using cleaning to remove all neutral and negative optimizations.

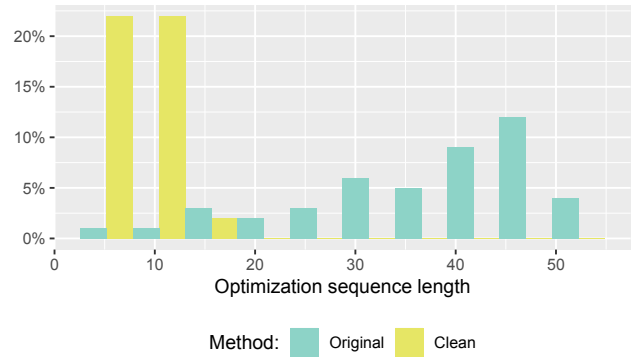

Fig. 10. Length of best optimization sequence per benchmark across methods originally and after cleaning.

Comparing the length of the best sequence before and after cleaning, we can observe that a significant amount of optimization passes – in the best sequences – are not effectively useful. Based on this observation one might want to use short sequences to find the best results. We explored this direction, but yielded results 10% worse than random. We expect that this is due to the fact that using a large sequence increases the chances to have an optimization with a positive influence on the estimated WCET.

3) Sensitivity to benchmark: As illustrated in the previous section, iterative compilation brings significant improvements to estimated WCETs. One would be tempted to derive general (benchmark agnostic) rules to improve WCET estimates, without going through the lengthy process of iterative compilation. In this section we investigate this opportunity in terms of sequence, optimization and finally learnt model.

Figure 11 presents the best estimated WCET for each benchmark using the best optimization sequences from other benchmarks, as they were acquired in the previous section. To evaluate the specialization of sequences to a given benchmark we use a leave-one-out cross validation, whereby each sequence is used to estimate the WCET for all benchmarks except the one it was found with. Using cross-validation allows us to check if the sequence brings improvement specifically to this benchmark, or if it can considered general across multiple benchmarks.

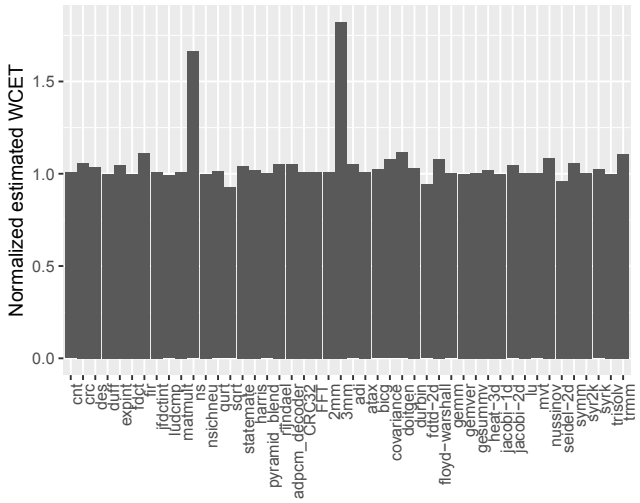

Fig. 11. Best estimated WCET per benchmark using the best optimization sequence from all benchmarks except the benchmark evaluated, normalized to the best estimated WCET for the benchmark

The geometric mean of the best estimated WCET for the cross validation is 0.823, 4% worse than the geometric mean of the best estimated WCET across all methods of 0.786. In terms of ranking, cross-validation gets 20% of the best results. This is only slightly better than random, which confirms our hypothesis that sequences are tailored to specific benchmarks.

Figure 12 is based on the best sequence for each benchmark across all methods, and presents the percentage of sequences containing a given optimization pass, after cleaning was applied to remove neutral and negative optimizations.

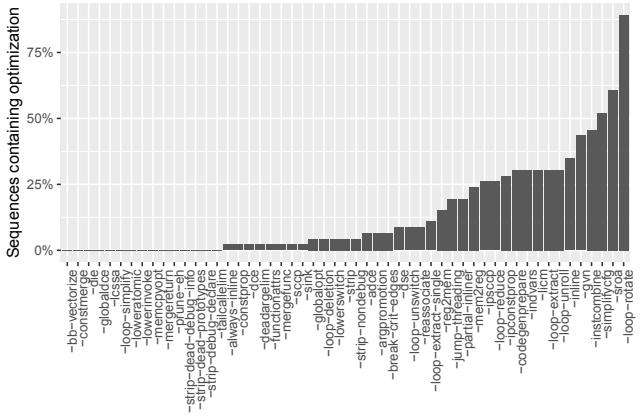

Fig. 12. Percentage of best sequences overall containing an optimization after cleaning.

We observe that best results are brought by only a subset of optimizations, with *-loop-rotate*, *-sroa* and *-simplifycfg* being present in more than 50% of the best sequences. On the contrary, we see that many optimizations are never present in best results, which hints at their neutral or negative effect on WCET estimates.

Figure 13 presents the percentage of neutral characterizations for each optimization. The presented characterizations are based on the 1000 sequences per benchmark, obtained using the cleaning method, and averaged across benchmarks using an arithmetic mean. The arithmetic mean was used due to the characterization similar nature and range. The optimizations are presented in the same order as in Figure 12, where the percentage of their presence in best sequences was illustrated. This ordering is used to relate the neutral characterization with the final best result.

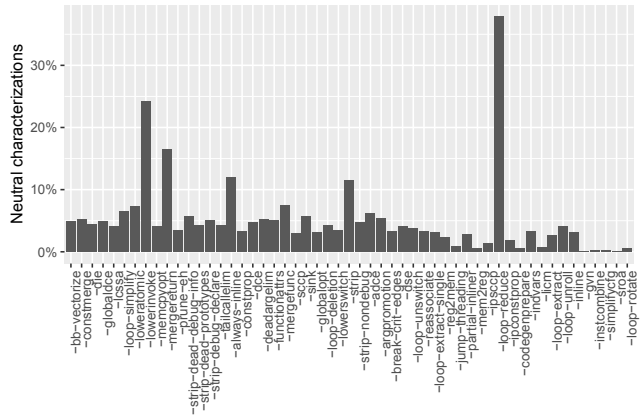

Fig. 13. Percentage of neutral characterizations per optimization across all benchmarks based on cleaning results.

We observe a small inverse correlation between the neutral characterization and its presence in the final result. This confirms that optimizations with neutral characterizations are on average not providing benefit to the best result. One exception is optimization *-loop-reduce* which has the highest neutral characterization, but is still present in 25% of best sequences. It may be caused by the fact that the optimization by itself is providing no gain (i.e. neutral), but combined with another optimization provides significant improvement.

V. Selection of the aiT's virtual unrolling factor

The aiT timing analysis tool uses a parameter named *virtual unrolling factor* to trade-off WCET estimation accuracy with analysis complexity (Section V-A). The automatic selection of this parameter in our proposal is presented in Section V-B.

A. Virtual unrolling in aiT and its impact on WCET estimates

Analysis of routines in aiT is achieved per context, e.g. a calling context for a routine with specific parameter values. Those contexts are used to restrict the possible values for each register. Contexts are used for all analyses (value analysis, loop bound analysis, cache analysis, etc).

The *virtual unrolling factor* option defines the number of contexts used to analyze loops. This concept is similar to *loop peeling* in compilers, which extracts the first N iterations of a loop. Unrolling is *virtual*, that is no code transformations are performed, and unrolling is only used in the timing analysis. In aiT, higher virtual unrolling translates into more loop contexts, with a different context computed for each of the first N iterations of the loop. Virtual unrolling is essential for loop bound analysis: a low virtual unrolling factor may result in a failure in finding the loop bound of some loop and, thus, in failure of the whole timing analysis. A high unrolling factor in contrast improves the precision of value analysis and consequently the quality of estimated WCETs.

The downside of a more accurate analysis is a longer analysis time due to the additional contexts. This extra time does not necessarily increase linearly with the virtual unrolling factor. In the case of kernels with two nested loops, virtually unrolling the outer loop creates multiple contexts, for each of which the inner loop is also virtually unrolled. Image processing kernels use triple nested loops, which could create a cubic complexity for virtual unrolling. This is an acute problem in iterative compilation in which WCET estimation has to be done for a large number of optimization sequences for each application.

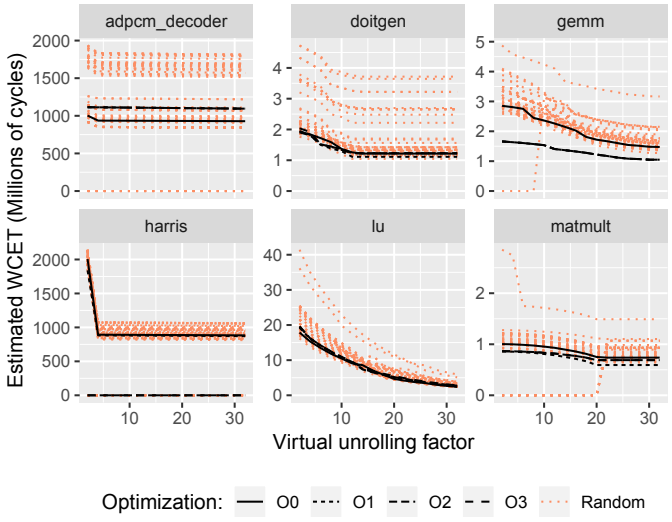

Fig. 14. WCET estimation of standard optimizations and 100 random generated sequences for virtual unrolling factor from 2 to 32.

To investigate this influence we ran WCET estimations for code compiled with standard optimizations levels (*O0* to *O3*) for different unrolling factors. Results for six of the selected benchmarks are plotted in Figure 14 for the estimated WCET, and Figure 15 for the runtime of the analysis. Analysis failures are represented with an estimated WCET equal to -1. We observe that increasing the unrolling factor improves the estimated WCET for some benchmarks (e.g. *gemm*, *lu*), but may also provide no noticeable improvement on larger virtual unrolling factors (e.g. *harris*, *matmult*). In terms of runtime, increasing the virtual unrolling factor increases the analysis runtime, whether there is improvement in the estimated WCET (e.g. *gemm*, *lu*) or not (e.g. *adpcm_decoder, harris*). In some cases the virtual unrolling may completely unroll the loop (e.g. *doitgen*, *matmult*), after which there is neither gain in estimated WCET nor additional runtime.

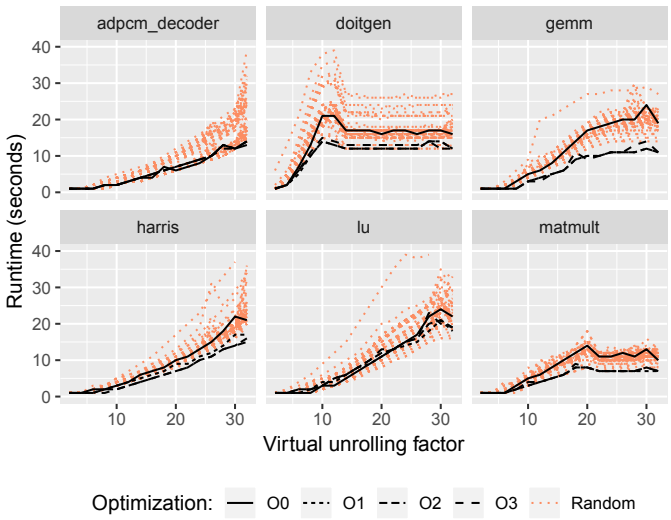

Fig. 15. WCET estimation runtime of standard optimizations and 100 random generated sequences for virtual unrolling factor from 2 to 32.

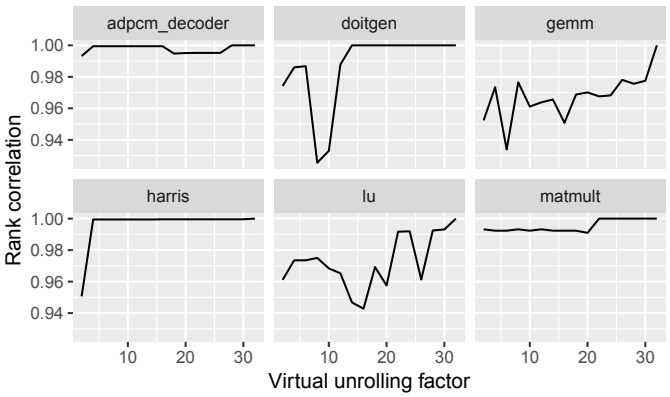

Fig. 16. Correlation of the ranking of estimated WCET for 100 random generated sequences at each virtual unrolling, with virtual unrolling 32 as reference.

B. Selection of virtual unrolling factor

Our objective is to maximize the *quality* of the estimated WCET, by selecting an unrolling factor tailored to each specific program, with a small overhead for its selection. We define this quality in two ways:

- Tightness: the estimated WCET is as close as possible to the actual WCET.
- Fidelity: the ordering of estimated WCETs for any pair of optimization sequences matches the one of their actual WCETs. For example, if $WCET_{ABC} < WCET_{BCD}$ for estimated WCETs at virtual unrolling factor 2 and $WCET_{ABC} > WCET_{BCD}$ for actual WCETs, virtual unrolling factor 2 has a low fidelity.

Validating both criteria would require a WCET oracle which is, in practice, impossible to acquire. For this purpose, we use the estimated WCET with the largest virtual unrolling factor as an indicator of the actual WCET.

Referring back to Figure 14, we have separated the results for the standard optimizations and 100 random sequences repeated for each virtual unrolling factor. For tightness, we observe that the standard optimizations and random sequences follow the same trend for each benchmark. An improvement in estimated WCET is visible on the standard optimization levels, which means standard optimization levels are a good indicator for a potential gain to be obtained by increasing the virtual unrolling.

In order to evaluate the fidelity of the estimated WCET we compute the rank correlation at each virtual unrolling factor using Kendall's τ correlation coefficient [16]. Rank correlation is a way to evaluate the fidelity of a ranking compared to a reference ranking. It is used in design space exploration to evaluate the fidelity of a model compared to the actual system. In this work we use it to evaluate the quality of the ordering at a given unrolling factor compared to our reference, a virtual unrolling of 32. Results of this analysis are presented in Figure 16, a coefficient of 1 being a perfect ranking. We observe that we reach good rankings when the estimated WCET reaches a plateau in Figure 14. This means that an absence of variation in the estimated WCET is an indicator that a given unrolling factor provides a high fidelity.

Algorithm 1 Selection of best virtual unrolling factor for a given benchmark.

```
for optim = O0 to O3 do
  prevWCET ← 1
  for unroll = 2 to 32 do
    estWCET ← aitT(benchmark, optim, unroll)
    if estWCET < θ × prevWCET then
      bestUnroll ← max(bestUnroll, unroll)
    end if
    prevWCET ← estWCET
  end for
end for
return bestUnroll
```

Based on the previous observations, we propose in Algorithm 1 the selection of the unrolling factor for a program. For efficiency considerations, we rely on standard optimizations as an indicator to select the unrolling factor. In addition, we limit the unrolling factor to the largest one that improves the estimated WCET, i.e. when reaching a plateau. Finally, we limit the unrolling factors explored to $\{2, 4, 8, 16, 32\}$. The threshold θ to consider an improvement is set to 0.9, meaning a 10% improvement. To further speed up exploration, an additional condition is added to only consider WCET estimations with a runtime smaller or equal than 30 seconds. This limits the virtual unrolling for benchmarks *adi*, *fdtd-2d*, *FFT, floyd-warshall, heat-3d, jacobi-2d, nussinov, harris, pyramid_blend* and *seidel-2d*. The selected virtual unrolling factor for each benchmark is presented in Table IV.

VI. Related work

Information on the flow of control of applications improves the tightness of WCET estimates. Beyond loop bounds,

TABLE IV
SELECTED VIRTUAL UNROLL FOR EVERY BENCHMARK.

2	cnt, des, fdct, fir, jfdctint, nsichneu, qurt, sqrt, statemate, floyd-warshall, pyramid_blend, adpcm_decoder, CRC32, adi
4	harris, duff
8	crc, ludcmp, ns, heat-3d
16	doitgen, nussinov, FFT, fdtd-2d, jacobi-2d, seidel-2d
32	expint, matmult, covariance, gemm, gemver, gesummv, symm, syr2k, syrk, trmm, 2mm, 3mm, atax, bicg, mvt, durbin, lu, trisolv, rijndael, jacobi-1d

which are mandatory for WCET calculation, examples of flow information include infeasible paths, or other properties constraining the relative execution counts of program points. Flow information can be obtained via two basic methods: static analysis [17]–[20] or annotations added by the application developer [21].

Compiler optimizations are often not used when designing hard real-time systems. One reason is the increased difficulty of automatic flow fact extraction using static analysis techniques on highly optimized code as compared to non-optimized code. Another difficulty, arising when flow information is given through developer-provided annotations is to safely transform them along with code optimizations. Finally, the certification of optimizing compilers is known to be difficult because of the inherent complexity of optimizing compilers [22].

However, using optimizing compilers is key for improved performance, both in the average case [23] or in the worst case [24], [25]. Several attempts have been made in the past to reconcile compiler optimizations and WCET estimation (that require safe and precise estimation of flow information).

A first research direction is to design WCET-aware compiler optimizations, that are known to improve WCET estimates, and have a well-known impact on flow information. Examples of WCET-directed compiler optimizations aim at optimizing the worst-case behavior of the memory hierarchy [26], [27] branch predictors [28], or perform WCET-aware loop transformations [29]. In contrast to these approaches, we take benefit of the full set of compiler optimization from mainstream compilers, designed for average-case performance, and automatically select the ones that minimize WCET estimates. In addition, in contrast to most related work that perform optimizations along the worst-case execution path (WCEP) our technique is a black-box technique that does not use the WCEP, and therefore does not have to handle WCEP variations.

When using source-level flow annotations for WCET estimation, these annotations have to be transformed together with code transformations [24], [30]–[32]. From an engineering perspective, tracing flow information in a compiler, albeit open-source, is a time-consuming task and the flow fact traceability code has to be maintained and be consistent alongside the code of the optimizations.

A last class of approaches, is to rely on mainstream compilers and automatically select optimizations that minimize the WCET. Many research, surveyed in [4], has been devoted to the automatic selection of the best set of compiler optimizations and the best ordering of applied optimization passes to optimize average-case performance. Compared to this rich set of techniques (see for example [33], [34]) the metric we wish to optimize is worst-case performance, and not average-case performance. To our best knowledge, only two recent studies have focused automatic selection of compiler optimization for WCET optimization. The study presented in [35], uses feedback-directed compilation to optimize WCET estimates, by feeding the compiler with WCET-oriented profile data. In contrast to [35], our work does not require any support from the compiler. The work presented in [25] automatically selects compiler optimizations and their ordering to minimize WCETs, using random or genetic techniques. As compared to [25] that suggests a fine-grain approach (per loop, using outlining) our work selects the same optimization sequence for all files, to reduce the search space, and the technique used for optimization selection outperforms both random and genetic search algorithms. In addition, compared to these two works, our research also automatically selects the best parameters for WCET estimation (the *virtual unrolling* parameter of the aiT timing analysis tool, that allows to find tradeoffs between analysis precision and analysis run time).

VII. CONCLUSION

In this work, we have outlined the challenges raised when using mainstream compilers to improve the estimated WCETs of programs. In particular, the experimental evaluation, over 46 varied benchmarks using an industrial static WCET estimation tool, shows that optimizations are application-specific, have uncertain impact on estimated WCETs and often render timing analysis impossible. To address these challenges and obtain improved WCET estimates within reasonable time, we proposed a novel learning approach that characterizes the impact of individual optimization passes in order to construct an optimization sequence that minimizes the estimated WCET. Results show that WCET estimates are reduced on average by 20.3% using the proposed technique, as compared to the best compiler optimization level applicable.

As future work, we consider sequence construction upon blocks of passes to take advantage of optimization passes interaction, and a more efficient identification of neutral passes than the time-consuming cleaning. An open question we would like to answer is whether the effect of optimizations on the estimated WCET could be predicted based on features observed in the application source/binary code. This would enable reusability of the learnt models for other applications and, thus, speed-up the exploration. Such directions are inline with compiler optimization heuristics [36] and constitute a significant challenge.

ACKNOWLEDGMENT

This work was funded by European Union's Horizon 2020 research and innovation program under grant agreement No 688131, project Argo. The authors would like to warmly thank Kelig Lesourd for his work on the experimental evaluation, and the anonymous reviewers for their helpful comments on earlier drafts of this paper.

REFERENCES

[1] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem - overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 36:1–36:53, May 2008.

[2] Y. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 16, no. 12, pp. 1477–1487, 1997. [Online]. Available: https://doi.org/10.1109/43.664229

[3] Y. Chen, S. Fang, Y. Huang, L. Eeckhout, G. Fursin, O. Temam, and C. Wu, "Deconstructing iterative optimization," *TACO*, vol. 9, no. 3, pp. 21:1–21:30, 2012. [Online]. Available: http://doi.acm.org/10.1145/2355585.2355594

[4] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A survey on compiler autotuning using machine learning," *ACM Comput. Surv.*, vol. 51, no. 5, pp. 96:1–96:42, 2019. [Online]. Available: https://doi.org/10.1145/3197978

[5] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, San Jose, CA, USA, Mar 2004, pp. 75–88.

[6] "aiT:the industry standard for static timing analysis." [Online]. Available: http://www.absint.com/ait

[7] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '93, vol. 22, no. 2. ACM, 1993, pp. 207–216.

[8] L. G. A. Martins, R. Nobre, J. M. P. Cardoso, A. C. B. Delbem, and E. Marques, "Clustering-Based Selection for the Exploration of Compiler Optimization Sequences," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 1, pp. 8:1–8:28, Mar. 2016. [Online]. Available: http://doi.acm.org/10.1145/2883614

[9] "Gaisler bare-c cross-compiler system." [Online]. Available: https://www.gaisler.com/index.php/downloads/compilers

[10] C. Cullmann and F. Martin, "Data-Flow Based Detection of Loop Bounds," in *7th International Workshop on Worst-Case Execution Time Analysis (WCET'07)*, ser. OpenAccess Series in Informatics (OASIcs), C. Rochange, Ed., vol. 6. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2007. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2007/1193

[11] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET Benchmarks: Past, Present And Future," in *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, ser. OpenAccess Series in Informatics (OASIcs), vol. 15. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010, pp. 136–146. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2010/2833

[12] L.-N. Pouchet and T. Yuki, "PolyBench/C." [Online]. Available: http://web.cse.ohio-state.edu/ pouchet.2/software/polybench/

[13] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, Dec. 2001, pp. 3–14.

[14] R. T. Mullapudi, V. Vasista, and U. Bondhugula, "PolyMage: Automatic Optimization for Image Processing Pipelines," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 429–443. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694364

[15] G. V. Conroy, "*Handbook of genetic algorithms* by lawrence davis (ed.), chapman & hall, london, 1991, pp 385," *Knowledge Eng. Review*, vol. 6, no. 4, pp. 363–365, 1991.

[16] H. Javaid, A. Ignjatovic, and S. Parameswaran, "Fidelity metrics for estimation models," in *2010 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov. 2010, pp. 1–8.

[17] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper, "Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution," in *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS 2006), 5-8 December 2006, Rio de Janeiro, Brazil*, 2006, pp. 57–66.

[18] M. D. Michiel, A. Bonenfant, H. Cassé, and P. Sainrat, "Static loop bound analysis of C programs based on flow analysis and abstract interpretation," in *The Fourteenth IEEE Internationl Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2008, Kaohisung, Taiwan, 25-27 August 2008, Proceedings*, 2008, pp. 161–166.

[19] S. Blazy, A. O. Maroneze, and D. Pichardie, "Formal verification of loop bound estimation for WCET analysis," in *Verified Software: Theories, Tools, Experiments - 5th International Conference, VSTTE 2013, Menlo Park, CA, USA, May 17-19, 2013, Revised Selected Papers*, 2013, pp. 281–303.

[20] J. Knoop, L. Kovács, and J. Zwirchmayr, "Replacing conjectures by positive knowledge: Inferring proven precise worst-case execution time bounds using symbolic execution," *J. Symb. Comput.*, vol. 80, pp. 101–124, 2017.

[21] R. Kirner, J. Knoop, A. Prantl, M. Schordan, and I. Wenzel, "WCET Analysis: The Annotation Language Challenge," in *7th International Workshop on Worst-Case Execution Time Analysis (WCET'07)*, ser. OpenAccess Series in Informatics (OASIcs), C. Rochange, Ed., vol. 6. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2007. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2007/1197

[22] X. Leroy, "Formal certification of a compiler back-end, or: programming a compiler with a proof assistant," in *33rd ACM symposium on Principles of Programming Languages*. ACM Press, 2006, pp. 42–54.

[23] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August, "Compiler optimization-space exploration," in *1st IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2003), 23-26 March 2003, San Francisco, CA, USA*, 2003, pp. 204–215.

[24] H. Li, I. Puaut, and E. Rohou, "Traceability of flow information: Reconciling compiler optimizations and wcet estimation," in *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, ser. RTNS '14. New York, NY, USA: ACM, 2014, pp. 97:97–97:106.

[25] I. Puaut, M. Dardaillon, C. Cullmann, G. Gebhard, and S. Derrien, "Fine-grain iterative compilation for wcet estimation," in *18th International Workshop on Worst-Case Execution Time Analysis (WCET'18)*, 2018.

[26] P. Lokuciejewski, H. Falk, and P. Marwedel, "Wcet-driven cache-based procedure positioning optimizations," in *20th Euromicro Conference on Real-Time Systems, ECRTS 2008, 2-4 July 2008, Prague, Czech Republic, Proceedings*, 2008, pp. 321–330.

[27] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen, "WCET centric data allocation to scratchpad memory," in *Proceedings of the 26th IEEE Real-Time Systems Symposium (RTSS 2005), 6-8 December 2005, Miami, FL, USA*, 2005, pp. 223–232.

[28] F. Bodin and I. Puaut, "A wcet-oriented static branch prediction scheme for real time systems," in *17th Euromicro Conference on Real-Time Systems (ECRTS 2005), 6-8 July 2005, Palma de Mallorca, Spain, Proceedings*, 2005, pp. 33–40.

[29] P. Lokuciejewski and P. Marwedel, "Combining worst-case timing models, loop unrolling, and static loop analysis for wcet minimization," in *The 21st Euromicro Conference on Real-Time Systems (ECRTS)*, Dublin / Ireland, jul 2009, pp. 35–44.

[30] J. Engblom, A. Ermedahl, and P. Altenbernd, "Facilitating worst-case execution times analysis for optimized code," in *10th Euromicro Conference on Real-Time Systems (ECRTS 1998), 17-19 June 1998, berlin, Germany, Proceedings*, 1998, pp. 146–153.

[31] R. Kirner, P. P. Puschner, and A. Prantl, "Transforming flow information during code optimization for timing analysis," *Real-Time Systems*, vol. 45, no. 1-2, pp. 72–105, 2010. [Online]. Available: https://doi.org/10.1007/s11241-010-9091-8

[32] P. Lokuciejewski, S. Plazar, H. Falk, P. Marwedel, and L. Thiele, "Approximating pareto optimal compiler optimization sequences - a trade-off between wcet, ACET and code size," *Softw., Pract.*

Exper., vol. 41, no. 12, pp. 1437–1458, 2011. [Online]. Available: https://doi.org/10.1002/spe.1079

[33] K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman, "ACME: adaptive compilation made efficient," in *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05), Chicago, Illinois, USA, June 15-17, 2005*, 2005, pp. 69–77.

[34] F. V. Agakov, E. V. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams, "Using machine learning to focus iterative optimization," in *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2006, pp. 295–305.

[35] M. Becker and S. Chakraborty, "Optimizing worst-case execution times using mainstream compilers," in *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems, SCOPES 2018, Sankt Goar, Germany, May 28-30, 2018*, 2018, pp. 10–13.

[36] S. Kulkarni, J. Cavazos, C. Wimmer, and D. Simon, "Automatic construction of inlining heuristics using machine learning," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Feb. 2013, pp. 1–12.